Using Incremental Planning to Foster Application Framework Reuse

Alvaro Ortigosa[†*] and Marcelo Campo[*]

[*]Universidad Nacional del Centro Prov. Bs. As. - Fac. Ciencias Exactas
ISISTAN Research Institute - Objects and Visualization Group

[†] Universidad Autónoma de Madrid, E.S.T. de Ingeniería Informática

E-mail: {ortigosa, mcampo}@exa.unicen.edu.ar

Contact author:

Alvaro Ortigosa
ISISTAN – Facultad de Ciencias Exactas – UNICEN
Campus Universitario, Paraje Arroyo Seco, (7000)
Tandil, Bs. As., Argentina

Abstract

In this work, we present an approach for documenting object-oriented application frameworks and using the documentation to guide the framework instantiation process. Our approach is based on a shift from a framework-centered to a functionality-centered documentation, through which a tool can guide the instantiation process according to the functionality required for the new application. The fundamental idea of our work is the combination of the concept of user-tasks modeling and least commitment planning methods to guide the instantiation process. Based on these techniques, the tool is able to present the different high level activities that can be carried out when creating a new application from a framework to the developer, taking as a basis the documentation provided by the designer through instantiation rules.

Keywords: Application Frameworks Reuse, Planning, Software Development Support Tools

1. Introduction

Object-oriented application frameworks constitute a great improvement in software reuse because they promote not only the reuse of single building blocks, but also the reuse of the design of systems or subsystems. A framework embodies design decisions like the application control flow, distribution of responsibilities among classes and communication protocols for implementing collaborations, which are implicitly reused by a user when developing a new application using the framework.

Depending on framework complexity, however, this development may be a hard and time-consuming task for novice users. Usually, in order to take full advantage of framework capabilities, a user must understand the internal details of the framework design. This involves, for example, the relationships among the different framework components and what is expected from the application specific code. Besides, considering that a framework is generally a very flexible design, this usually implies a design harder to understand.

These reasons make good quality documentation an essential factor to make a framework successful. However, traditional design and code documentation techniques are not enough to describe the complexity of a framework, specially if considered the different kinds of users that may need to access framework documentation [5]. Butler et al. [1] describe four kinds of framework (re)users: application developer, framework maintainer, developer of another framework and verifier. Taking into account this variety, different documentation methods have been proposed for documenting frameworks. Some of them are informal and prescriptive [5][13][17], that is, they describe how the framework should be used. Some other methods are more formal and descriptive: they describe the framework design, and the user has to deduce how to use it [4][18]. Every technique is oriented to a given kind of framework user, and even some of these approaches are able to provide good descriptions of some framework aspects, none of them can successfully satisfy all the framework documentation requirements [16].

In this context, more powerful tools to support the production of good documentation and guidance for framework users become increasingly important. In this work we present a new approach to support framework documentation in such a way it can act as an active guide for the tasks that users must accomplish to build a specific application using the framework. The fundamental idea of our work is the combination of the concept of user-tasks modeling and least commitment planning methods [19] to guide an instantiation process. Based on this technique, a tool can present the developer the different high level activities that can be carried out when creating a new application from a framework, taking as a basis the documentation provided by the designer through instantiation rules. For example, if the framework is on the accounting domain, some of the initial activities can be to create a new type of account, or to describe a new algorithm to calculate the tax rate. For each of these high level activities, there is a list of tasks that the user must carry out in order to complete the activity. When the user selects her next objective, the tool is able to build the sequence of tasks that have to be done to accomplish that objective; this list of tasks is called the instantiation plan, and the process of plan creation is named planning. For this end a specialized planning algorithm, called *PIT* (Planning Instantiation Tasks), was developed. In this paper we present the main characteristics of the planning approach and a short example of the instantiation tool developed to support the approach.

The paper is organized as follows. Next section discusses framework documentation requirements and the role of user tasks applied to the framework instantiation process. In section 3 we present the concept of incremental planning of instantiation tasks, introducing a representation for the needed framework documentation. Section 4 presents *SmartBooks*, a method for framework documentation based on those ideas, and *HiFi*, a tool developed to support the approach, along with an example of its application. In section 5, the *PIT* algorithm is described. Finally, in section 6, the main conclusions of the work are presented.

2. Framework Documentation and User Modeling

It is widely accepted that good framework documentation must combine diverse methods for documenting frameworks [14]. Focusing on an application developer user (that is, one that uses the framework for developing a new application), an important technology is represented by the so-called active cookbooks [17]. A cookbook contains a set of recipes, where each recipe describes, in an informal way, the solution for a specific problem using the framework. Recipes do not explain the rationale of the design or why a problem must be resolved in a given way, but just explain how the problem can be solved using the framework. Active cookbooks are tools that enact recipe descriptions, providing the user an interactive interface that guides her through the instantiation process. One of the reasons of cookbook success is that humans are good at following step-by-step directions.

Cookbooks are a promising approach, but they are limited by two fundamental drawbacks: firstly, the lack of adequate environments that support the creation of the cookbooks. Currently, cookbooks must be created from scratch, and all the work is responsibility of the documentation writer (consistency rules, if any, the steps to be done, etc). Secondly, the recipes and cookbooks present the problem of little flexibility. The more detailed the assistance provided by a given recipe is, the less possible it is for the user to follow an instantiation process different from the one stated by the recipe. This problem is more evident when dealing with active cookbooks, where the user has to follow the embedded recipe up to the last detail, or must resign herself to work without the tool. Even worse, framework documentation should address the needs of developers with varying levels of experience with the framework, giving each one the opportunity to interact with the documentation at the level she prefers.

Besides these drawbacks, the focus of the documentation represents a critical point. In general framework designers tend to explain how to use a framework taking into account what we call a framework-centric view. That is, the documentation tends to explain how to use or add specific components instead of centering the specification around what functional requirements are satisfied by a given combination of framework components or methods.

Under this view, we claim that a more powerful tool should provide the user with some mechanism to express the requirements that her specific application should fulfill, and provide guidance about what programming activities should be done in order to get such behaviors with the framework. On the other side, a documentation support tool should emphasize the organization of the documentation regarding this aspect as a central one.

Seeing the instantiation process as a set of programming activities (i.e., subclassing, method overriding, etc.) then such activities can be assimilated to the concept of user-tasks, successfully used in the modeling of interactive applications [6]. In this way, the process of producing instantiation documentation can be seen as a process of user-tasks modeling, as we explain below.

2.1. Instantiation Tasks

During the last years there has been a growing effort devoted to the modeling of the tasks a user can accomplish while working with an interactive application [6][12]. The use of task models in complex applications allows a richer interaction between the user and the system, since the system is able to more precisely understand the user objectives, and give her support towards their accomplishment. This support can be in terms of user modeling and adaptation of the application to her needs, support for the development of help systems for applications [11], support for the use of multiple connected applications, etc. One of the roles of task models is to connect the semantics of user actions to the lower level of elementary actions (events) imposed by window toolkits. Usually, the applications under consideration have a well defined set of basic user actions, which is still too complex for a novice user to comprehend or for a designer to keep in mind while thinking about other aspects of the interface.

The instantiation of software frameworks is an activity that is also based on a well defined amount of basic tasks, like for example class specialization and method overwriting, but it presents a higher degree of complexity. A framework documentation tool should provide means to allow a designer to describe the different ways a framework can be used, by describing the different *instantiation tasks* (i.e., subclassing, method implementation, etc.) that are needed to obtain a given functionality with the framework, along with the conditions under which such tasks can be executed. Nowadays, standard task management systems consist essentially of an event parser that is able to match sequences of user events against task models in order to decide which tasks the user is accomplishing, and also to match sequences of higher level tasks against higher level task models. A task management system that is suitable for the control of framework instantiation processes must also include a dynamic basis of pending tasks, and a rule interpreter that modifies the set of pending the tasks according to the actions of the user. These modifications can include the addition of new pending tasks that are necessary to accomplish after some others, or the subtler modifications needed when a task like the creation of a software component is undone.

Having the information about which tasks should be done in order to implement some functionality, it would be possible to generate, at least, a partial sequence of such tasks that guide the user in the instantiation process. This sequence of tasks can be seen as an instantiation plan that can be produced through some planning algorithm.

Planning is an AI technique that, given a goal and a set of possible actions, produces an execution sequence to reach the goal. In this case the goal is to build an application that must satisfy several functional requirements and the development is done through the adequate combination of several instantiation tasks that produces the final application, or at least, the main parts that can be built using the framework.

The instantiation tool is based on a planner that, given the functionality the user wants or needs, produces an instantiation plan. This plan is a set of partially ordered tasks that must be executed in order to obtain the desired functionality. Some tasks can be automatically executed and some others must be carried out by the user. An example of automatic task is the reuse of an existing component. Example of user task is the creation of a new class, or the redefinition of a given method.

It is necessary to make clear that, in order to be practical, a tool like this has to allow a flexible execution of tasks. It has to allow tasks to be achieved in any order, to be interrupted in order to work on other tasks, and even to be cancelled at any moment, and the system must be able to adapt itself to the new situation risen at each of these steps. In this sense the use of least commitment planning techniques [19] appears as an interesting alternative to build partial plan sequences in a flexible way. Next section presents the general format of the instantiation rules that must be provided to the planner as input. In section 5 the *PIT* algorithm, specially developed to support the planning of instantiation tasks based on instantiation rules, is introduced.

3. Incremental Planning of Instantiation Tasks

The *PIT* algorithm is an adaptation of the *UCPOP* planning algorithm [19], specially developed to manage the requirements of a framework instantiation process. The *UCPOP* algorithm is further extended to support not only partial planning, but incremental planning too.

The planning is said to be partial because the resulting tasks are just partially ordered. Given a pair of tasks, they can have a specific ordering (that is, one should be finished before the other can be executed) or no ordering, so any one can be executed first, and even they can be executed in parallel.

Besides being partial, the planning is incremental. That is, the instantiation plan is not generated at once, but it is done gradually, according to the tasks executed by the user. Sometimes the plan generation is halted, waiting for some user input. Moreover, the user can incrementally redefine her objectives, or modify past decisions, and the plan generation must be updated accordingly.

3.1. Instantiation Rules

The input of the planner is a set of rules that describes the necessary steps to obtain the desired functionality. These rules are generated by the tool through a graphical interface that allows the designer to express instantiation actions along with general design documentation of the framework. The tool is described in the next section.

Some of these rules are framework specific, while others describe general situations of framework instantiation. The general form of a rule is:

```
precondition list → effect
```

and represents changes on the software and/or the plan state.

While implicit in this representation there is an instantiation action (that, executed when the preconditions hold, will produce the effect), only preconditions and effect are significant to the planning algorithm. Specifically, effect states a condition that will be true when the preconditions are true. So, in every step the planner tries to make true the preconditions of an action whose effect is part of the goals.

The following rules are specific for a visualization framework named *LuthierAbstractors*[2], which provides support for building visualizations with abstraction scales and filtering. All the examples of this paper are based on the *LuthierAbstractors* framework because, due to space constraints, they should be as simple as possible. Because of the same reasons, the framework cannot be explained here; see [2] for a detailed explanation.

```
getUserInput(‘Items to Filter ?’, Items), implements(‘filter’, Items) → functionality(‘Management of Visualization Focus’)   (1)
tryUseComponent(Component, TransportFilter) → implements(‘filter’, ‘[buses, subways, train]’)                                (2)
tryUseComponent(Component, ZoneFilter) → implements(‘filter’, ‘[residential, commercial, industrial]’)                       (3)
tryUseComponent(Component, FilterAbstractor) → implements(‘filter’, Items)                                                   (4)
```

Rule (1) states that in order to obtain visualizations with management of the visualization focus, a component (or set of components) implementing a ‘filter’ can be used. Besides, the component must be selected taking into account the sets of items the user wants to filter. Rules (2) and (3) represent that the *TransportFilter* or *ZoneFilter* components can be used to filter some specific items. Finally, rule (4) provides a component to be used by default, when there is not a more specific component.

It must be noticed that the terms used to express the functionality ('Management of Visualization Focus' in the example) are arbitrarily fixed by the designer. In this way, the tool must show the user the different functionality implemented by the framework, so the user can choose. This representation can be further refined if a domain language were defined to provide a textual vehicle to express the functionality desired for a specific application, although this aspect is beyond our current goals.

Examples of generic rules are rules that state how a component can be used:

```
exists (class (CompDesc, X)), choose (CompDesc, ['refine', 'reuse'], Answer), useComponent (C, CompDesc, Answer)
        → tryUseComponent (C, CompDesc).                                                                        (5)
¬ exists (class (CompDesc,X)), defineNewComp (C,CompDesc) → tryUseComponent (C, CompDesc)                       (6)
```

The first rule states that, in order to use an existing component, the user can choose between using it "as is", or creating a specialization of it. The second one shows what to do if the component does not exist. These framework independent rules are called "primitives", because they are the building blocks to describe the framework specific rules.

Some primitive rules have, as a side effect, the creation of tasks that must be carried out by the user. For example, if the planner finds that the implementation of a given functionality requires the specialization of a component and the redefinition of some methods, the corresponding tasks are created and queued as "pending" tasks for the user. For example, the following primitive rule describes how to instantiate a new component:

```
pendingTask ('DefineNewClass', FuncDescs) → implementComponent (C, FuncDescs)                                   (7)
```

There exists another type of task, called "waiting task", which represents a task whose result is necessary for the planning process itself. When the planning algorithm finds a waiting task as a precondition of a desired post-condition, it creates the task and is suspended. Once the user completes the task, the algorithm is reinitiated. One of the primitive rules used to get input from the user is

```
waitingTask ('GetInputTask', Description, Return) → getUserInput (Description,Return)                          (8)
```

This rule representation is used by the *SmartBooks* method to assist the framework instantiation process. In order to support the approach, it was developed a planning algorithm that uses these rules as input. With the goal of making simpler the understanding of these ideas, firstly the method is introduced in the next section, and then an explanation of the *PIT* algorithm is presented in section 5.

4. *SmartBooks*

SmartBooks is a method for helping in framework instantiation through intelligent documentation. The method is based on the idea of generating an instantiation plan from the description of the functionality required for a given application. Essentially, the method prescribes that the framework should be documented using documentation books. These books, different from the cookbooks, contain not only descriptions of the reuse situations anticipated by the framework designer; they also contain documentation of the framework design: component interfaces, responsibilities, communication protocols. That documentation can include, among other things: the meta-patterns and design patterns implemented in the framework, contracts, pre and post conditions and any other information the developer can provide to describe the design of the framework and the conditions that have to be maintained by the classes developed by the framework users. Additionally, the designer must provide information about how the framework can be used to satisfy different functional requirements in the form of instantiation rules. From this information, and specially from the instantiation rules, the planning algorithm can produce a sequence of instantiation tasks that must be carried out by the developer, depending on the functionality desired for the new application.

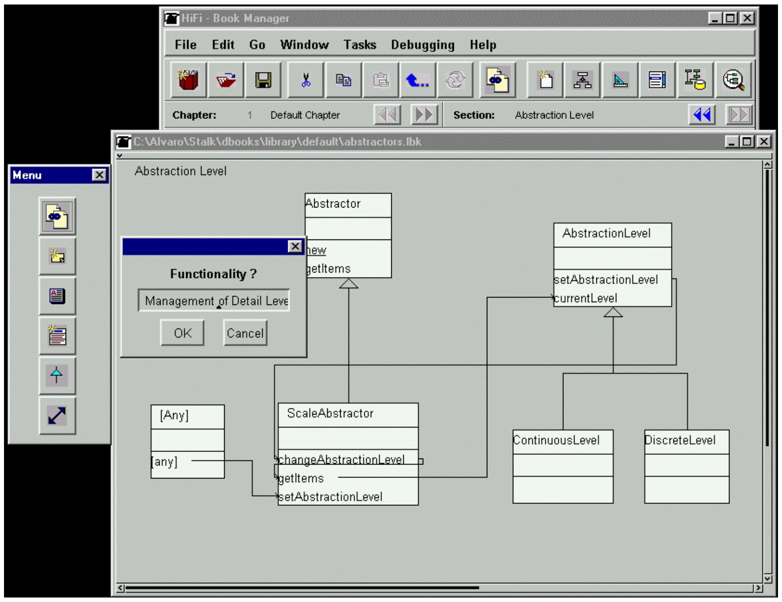
Based on the *SmartBooks* method it was built a prototype tool, called *HiFi* (Helping in Framework Instantiation). This tool, implemented in *Smalltalk*, provides a seamless environment to produce documentation about a framework and the application themselves. New documentation can be added during the development process, either about the framework or the application being implemented, and the new documentation can be used to validate the work already done and generate more accurate plans for the rest of the instantiation.

The tool is also fully integrated with the *Smalltalk* browser in such a way that it can monitor the user actions and reacts accordingly. For example, if the user produces code that is related to some instantiation task involved in the instantiation plan, *HiFi* detects the changes and the plan can be updated. In the same way, the plan can be augmented with tasks needed to keep the software developed consistent with the documented design. These control activities are carried out by an intelligent agent, based on consistency rules. While this paper is focused on the generation of the instantiation plan, the functionality of this agent is out of its scope, and it is further explained in [9].

The next two sections present an example of the application of *SmartBooks* through *HiFi* to document and instantiate the *LuthierAbstractors* framework. The use of the tool consists of two separated processes: in a first step, the framework developer uses the tool to create the documentation, and then the application developer uses the documentation to create an instance of the framework.

4.1. Documenting a Framework

On the documentation stage, the framework developer must describe the design of the framework using UML notations [15], being this documentation structured as a design book. In this book the user can navigate through the different diagrams, specifications, textual annotations and the framework code. Besides traditional notations, the designer should describe what functionality could be implemented using the framework, and how the functionality is related to the framework components (instantiation rules).



Functionality ?

Management of Detail Leve

Fig. 1 – Example of Functionality Specification

Figure 1 shows an example of describing the functionality implemented by a collaborative group. In this example, the designer documents a collaborative group (called *Abstraction Level*), and specifies that this collaboration implements the “Management of Detail Level” functionality. The specification describes the components needed for implementing this functionality and imposes some constraints on these components.

Specifically, the description shown in figure1 represents:

1. For implementing the “Management of Detail Levels” functionality, an instance of *ScaleAbstractor* (or one of its subclasses) and an instance of a subclass of *AbstractionLevel* should be used.
2. The method *setAbstractionLevel* of *ScaleAbstractor* must be called from another object (i.e., the caller class and method are not defined)
3. The *setAbstractionLevel* method of *AbstractionLevel* must call the *changeAbstractionLevel* of *ScaleAbstractor*
4. The *changeAbstractionLevel* method of *ScaleAbstractor* must send a *getItems* message to self.
5. The *getItems* method of *ScaleAbstractor* must invoke the method *currentLevel* of *AbstractionLevel*.

These descriptions are used, mainly, to generate the rules used by the planner to generate the instantiation plan. For example, from the description of figure 1 rule (9) is derived. This rule is added to the other rules defined for the framework and the generic rules used independently from the framework.

```
tryUseComponent(X1, ‘ScaleAbstractor’), tryUseComponent(X2, ‘AbstractionLevel’), calls([Any],[Any],X1,’setAbstractionLevel’),
calls(X2,’setAbstractionLevel’,X1,’changeAbstractionLevel’), calls(X1,’changeAbstractionLevel,X1,’getItems’),
calls(X1,’getItems’,X2,’currentLevel’) → functionality(‘Management of Detail Levels’).                                    (9)
```

Functionality descriptions are not only used for rule generation, but also they are linked with the rest of the documentation, and can be navigated by the framework user when trying to understand how the framework works.

It must be noted that there exist subclasses of *ScaleAbstractor* (which are not shown in figure 1), that redefine the *functionality* attribute of the *ScaleAbstractor* class, by defining specific scales managed by the subclasses. Because of this refinement, the planner will have to ask the scale to be managed, in order to be able to choose the right component.

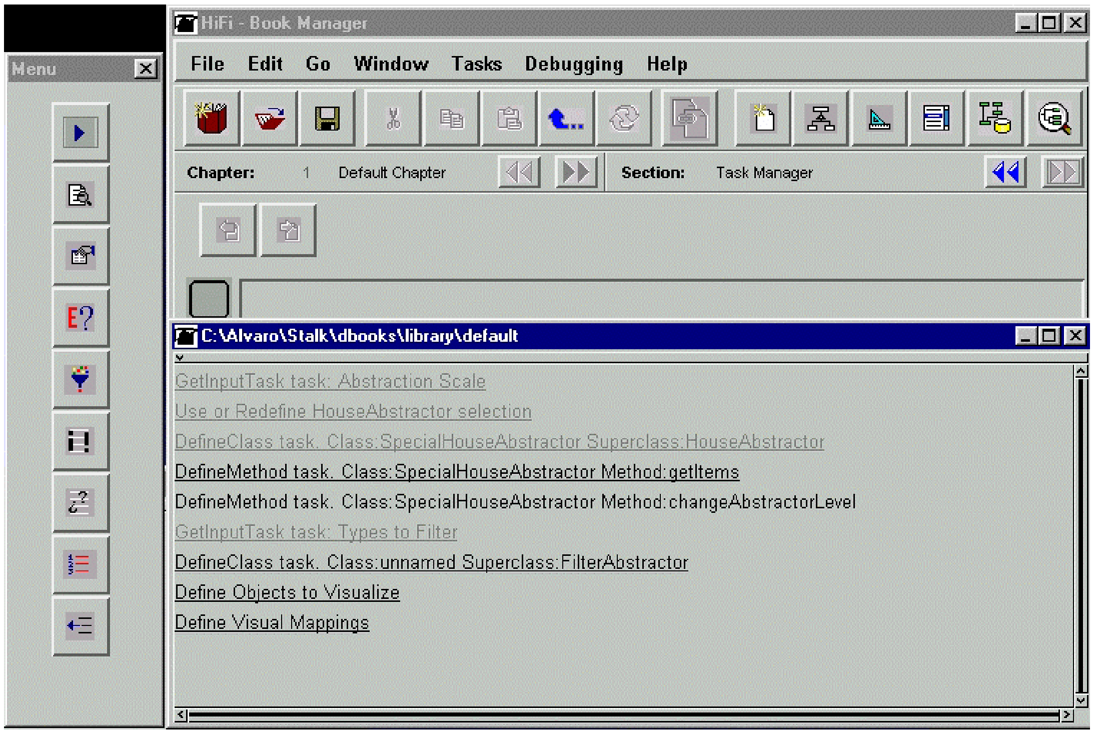
4.2. Planning and Instantiating

Once the framework has been documented, *SmartBooks* can be used to assist the implementation of applications using the framework. Based on the designer description, the tool shows the user the functionality provided by the framework. The user selects the functionality required for the application to be implemented, and the instantiation plan is generated, according to the describing rules[1].

In the example, the user selects that the application should provide management of detail levels and visualization focus. After this selection, the planning algorithm is called with the following goals:

```
functionality (‘Management of Detail Levels’), functionality(‘Management of Visualization Focus’)
```

Figure 2 shows the user interface of the task manager, that is, the list of tasks to be executed by the user generated by the planner. Through this interface the user can, for example, select a task to be executed, inspect a task, or access the documentation that explains the context of the task (why it should be executed, constrains, etc.).


Menu
HiFi - Book Manager
File Edit Go Window Tasks Debugging Help
Chapter: 1 Default Chapter
Section: Task Manager
C:\Alvaro\Stalk\dbooks\library\default

GetInputTask task: Abstraction Scale
Use or Redefine HouseAbstractor selection
DefineClass task. Class:SpecialHouseAbstractor Superclass:HouseAbstractor
DefineMethod task. Class:SpecialHouseAbstractor Method:getItems
DefineMethod task. Class:SpecialHouseAbstractor Method:changeAbstractorLevel
GetInputTask task: Types to Filter
DefineClass task. Class:unnamed Superclass:FilterAbstractor
Define Objects to Visualize
Define Visual Mappings

Fig. 2 - Task Manager interface. The tasks already executed are displayed in gray, and the pending tasks in black. Underlined tasks are required, and the others are optional

The planner starts to work with the first goal, *functionality (“Management of Detail Levels”)*, and finds rule (9). As a consequence, the action and its preconditions are added to the agenda. The next condition to be satisfied is:

```
tryUseComponent (X1, ‘ScaleAbstractor’)
```

At this point, the planner applies a primitive rule, that specifies how to use a component. From this rule, the planner finds it needs further information from the user, namely, the abstraction scale that must be managed. A task of type *GetInputTask* is generated with this goal, and based on the user input, the planner concludes that a *HouseAbstractor* component can be used.

Every time a component is selected to be used, the user can choose either to use it “as is”, or to create a specialization of the component. This fact is described by the primitive rule (5) that generates a task (second task of figure 2) so the user can choose. In the example, the user has selected to create a specialization of the component, so the planner generates a task to define a new subclass of *HouseAbstractor* (third task on the list). Besides, following the designer specification, two more tasks to define methods of the new class are generated (fourth and fifth tasks).

In the same way the second requirement is analyzed. In this case, the planner asks the user to provide the list of types of items to be filtered. With the user answer, the planner concludes that no existing component implements the required functionality, but according to rule (4) the *FilterAbstractor* component implements the generic *filter* functionality. For this reason, it generates a task to define a new class, subclass of *FilterAbstractor* (line seven of figure 2)

Besides the tasks for implementing the specific functionality, there are tasks that must be executed every time an application is built using the framework. In this case, it is necessary that the user defines the objects to be visualized and the presentations of these objects. This is represented by the last two tasks of the list.

5. *PIT*

A planning algorithm, named *PIT*, was developed to support the *SmartBooks* method. This algorithm was specifically designed to fulfill the requirements of the framework instantiation domain. *PIT* is, basically, a loop that tries combinations of goals. If a plan cannot be built for the complete set of goals, it tries with a subset of these goals. The algorithm works using backtracking, and eventually it will try every combination of goals until building a plan for a given subset or returning an empty plan. If a plan is built for a proper subset of the original goals, it means that the framework documentation is not enough to completely describe how to implement a given functionality, but user tasks for implementing the known parts are generated anyway.

PIT is called with three arguments: a plan (initially null), a set of goals to be satisfied, and a set of rules describing actions that can be used to satisfy the goals. The plan itself is composed by six elements: a list of instantiated actions; a list of partial orderings; a set of causal links; a list of variables with their corresponding values (bindings); a list of pending tasks created through the planning; and a list of goals which are not considered in the plan. The six of them are empty in the null (initial) plan.

Step 1 of *PIT* tests if the agenda is empty, and if true, returns the current plan. Just a part of this returned plan will be used directly by the user. The user interacts through the tasks produced during the planning process (the *tasks* component of the plan), but she does not need to know about the middle steps produced by the planner. The information can be used, anyway, to assist the user, explaining the context of the tasks she is required to execute. Besides, all the information about the resulting plan will be necessary if the algorithm must be executed again, for example after a modification on the user goals.

Step 2 calls the auxiliary function, which does most of the planning work. If *Aux_PIT* builds a plan, it is returned. Otherwise, step 3 tries different subsets of agenda, so that to find an instantiation plan for a given combination . Step 4 makes the recursive invocation with the new agenda. If no plan can be built for any subset of goals (except the empty one), an empty plan is returned. That means that the planner could not build a plan for any of the initial goals, either combined or in isolation.

In order to guarantee the algorithm halts and tries all the solutions before failing, the function that chooses a subset of goals to be removed must be carefully designed. It should consider orderly all the possible subsets. For a correct behavior of the planner, this function must consider first the smaller subsets. In this way, if it is possible to generate a plan, this plan will consider as much goals as possible.

Algorithm**: *PIT* (⟨A,O,L,B, tasks, pending⟩, agenda, Δ)**

1. Termination: If *agenda* is empty, return *(⟨A,O,L,B, tasks, pending⟩)*
2. Call auxiliary function: *return = PIT_Aux (⟨A,O,L,B, tasks⟩, agenda, Δ).* If it has success, return *return*. Otherwise, go to step 3.
3. Goal selection: choose a (no empty) set of goals $G_{pend} \subset$ *agenda*. If every proper subset of agenda was tried, return an empty plan. Otherwise, let agenda' = agenda $\cup$ pending - G_{pend}, pending' = G_{pend}.
4. Recursive call: return *PIT (⟨A,O,L,B, tasks, pending'⟩, agenda').*

The *Aux_PIT* function in every loop looks in the agenda for a goal to be satisfied, and if it finds an action describing how to satisfy it, adds its preconditions to the agenda.

Step 1 tests if the agenda is empty, and if true, returns the current plan.

Step 2 selects the next goal to be satisfied (⟨Q, A_c⟩), where Q are the preconditions of A_c. As the goal ordering can be significant, the first goal of the agenda is always selected. Steps 2.a and 2.b handle the cases of multiple preconditions. In 2.c, if the precondition is a *waitingTask*, the corresponding task is created, put in the queue of task to be executed, and the execution is halted (by calling to *nextEvent* procedure) until an event is produced. An event represents a task completed by the user. If the completed task is the one the planner is waiting for, the agenda is updated to reflect the result of the user action, and the algorithm is restarted.

If the precondition implies a pending task must be executed by the user, the task is added to the list (2.d). Next point, 2.e, handles the special operators: *if*, *no* and *forEach*. Finally, if the opposite precondition is also a precondition of A_c, a plan cannot be built.

Step 3 looks for an action that satisfies Q. With this purpose, it looks first for an action already instantiated that can be used (that is, it tries to find if the work was already done by a previous action). If one of these previous actions can be used, it is selected. In the case none of the instantiated actions is useful, the algorithm looks for an action that describes how to satisfy the precondition. It takes the first rule it finds, but if in the future fails to satisfy a goal, will backtrack to this point and a different rule (if it exists) will be selected. Then, it updates the plan representation. In the case there is no way to satisfy the current goal, the function backtracks to find alternative ways to satisfy the current agenda. If it does not find a way, the control is returned to *PIT*.

Step 4 updates the agenda and list of actions. If the action selected in 3 is an old action, it just needs to remove the current goal from the agenda. On the other side, if it is a new action, its preconditions are added to the agenda. It also updates the list of bindings and the list of actions.

Step 5 verifies if the new action needs some ordering relationship with the old ones, and in this case the necessary relationships are added to the partial ordering list.

Finally, step 6 makes the recursive invocation.

```
Function PIT_Aux: (⟨A,O,L,B, tasks⟩, agenda, Δ)

1. Termination: if agenda is empty, return {A, O, L, B, agenda, tasks}.
2. Goal reduction: remove a goal ⟨Q, $A_c$⟩ from agenda
a) If Q is a conjunction of $Q_i$ then put each <$Q_i$,$A_c$> in agenda; go back to step 2.
b) If Q is a disjunction of $Q_i$ , then choose the first $Q_k$ from the disjunction and put <$Q_k$,$A_c$> in agenda; go back to step 2.
c) If Q is a waitingTask(TaskClass, Description, Return), create and put in the queue the task so the user executes it.  Let result = NextEvent. After returning, $B' = B \cup \{(Return, result)\}$; A'= A; O'=O; L'=L; tasks '= tasks. Go to step 6.
d) If Q is a pendingTask, let tasks' = tasks $\cup$ {Q}. The rest of the variables stays without changes. Go to step 6.
e) If Q is an operator, call HandleOperator(Q, $A_c$, ⟨A, O, L, B⟩, agenda, ⟨A', O', L', B'⟩, agenda'). If the function does not fail, go to step 6 without changing any variable, except agenda'. Otherwise, fail.
f) If Q is a literal and a link $A_p \rightarrow^{\neg Q} A_c$ exists in L, then fail (the plan is impossible)
3. Selecting an action: select an action $A_p$ with effect R. Firstly try with the existing actions (A) and then try to apply an action from Δ. In any case, Ap < Ac must be consistent with O and the effect R (or consequent if the effect is conditional) unify with Q given B. If no action satisfies the requirements, then fail. Otherwise, make:
$L' = L \cup \{A_p \rightarrow^Q A_c\}$
$B' = B \cup \{(u,v) | (u,v) \in MGU(Q,R,B)$ ^ u,v are variables from R}
 $O' = O \cup \{A_p < A_c\}$.
4. Enable new actions and effects: let A'= A y  agenda' = agenda . If $A_p \notin A$ then add $A_p$ to A', add <preconditions($A_p$) \ MGU(Q,R,B),$A_p$> to agenda', add $\{A_0 < A_p < A_\infty\}$ to O and add the no-codesignation constraints ($A_p$) to B'. If the effect is conditional and it has not been used to establish a link in L, then add its antecedent to agenda after substituting with MGU(Q,R,B).
5. Causal link protection: for each causal link $l = A_i \rightarrow^P A_j$ in L and for each action $A_t$ which threatens l nondeterministically choose (or, if no choice exists, fail):
Demotion: add $A_t < A_i$ a O'
Promotion: add $A_j < A_t$ a O'.
Confrontation: if the $A_t$'s threatening effect is conditional with antecedent S and consequent R, then add <¬S\MGU(P,¬R) , $A_t$ > to agenda'.
6. Recursive call: If B is inconsistent then fail; otherwise call PIT_Aux(<A',O',L',B', tasks'>,agenda', Δ)
```

The *NextEvent* function halts the algorithm while waiting for the user executing a waiting task. Once the task is finished, its result is returned

```
Funtion NextEvent ()

1. Wait next event: stop until the user produces an event, by finishing a task. Let event = ⟨EventName, TaskDesc, Result⟩.
2. Restart planning: return Result
```

The *HandleOperator* function manages special operators, used to modify the normal behavior of the planner. The *if* operator is used to represent that a given action can be applied only if another condition, an argument, is already present in the plan. The *not* operator represents the opposite condition: the action can be applied only if a given condition was not made true yet. Finally, the *forEach* operator is used to express a condition that must be satisfied by a list of elements. This last operator is needed because the *PIT* algorithm does not provide universal quantifiers[2].

```
Function HandleOperator (Q, Ac, ⟨A, O, L, B⟩, agenda, ⟨A', O', L', B'⟩, agenda')
case(Q)
if (condition)       if (Ap, condition)$\notin$ A, fail. Otherwise, L' = L + {Ap→conditionAc}, agenda=agenda'.
not (condition)      if (Ap, condition) $\in$ A, fail. Otherwise, L' = L + {Ap→not(condition)Ac}, agenda=agenda'.
forEach(Var, List, Objective) agenda' = agenda. For each element X from List, put in agenda' a new goal equal to Objective, with every X Var replaced by X.
```

6. Conclusions

In this paper an approach to document application frameworks was introduced. In addition, a planning algorithm and a tool designed to support the approach were presented.

The main contribution of this work is to show how planning techniques can be used to support the framework instantiation process. The *SmartBooks* method proposes the combination of least commitment planning techniques and user-tasks modeling. The *HiFi* tool exemplifies that this approach is a good vehicle to implement functionality-oriented assistance. *SmartBooks* is especially useful for inexperienced users, but it is flexible enough to assist more expert users.

Even so the examples presented on this paper are simple, the approach has already been thoroughly tested with a medium sized framework, and the results show that the method can be scaled up [10]. The main reason is that, independently of the framework size and complexity, the instantiation tasks are basically subclassing and method implementation. This scalability is further supported by the capability of expressing implementation alternatives as independent instantiation rules, which enables the modularity of the documentation.

Starting from this point, there are several aspects of framework instantiation support that need more research. One of these aspects is team based development. In this case, the task manager can be extended in order to manage the distribution of instantiation tasks among the developers, and the coordination of these developers.

It is also necessary to consider how to assist when the framework design changes. If the implementation of a given functionality or the framework architecture is changed, documentation has to be updated accordingly. The documentation tool should provide assistance oriented towards those changes.

Regarding scalability, it should also be considered the behavior of the *PIT* algorithm as the number of rules and goals increases. Even so the time consumed by the algorithm grows exponentially with the input size, this time will never be significant compared with the time involved in a development process. In that way, it is only necessary to guarantee that the algorithm halts.

One of the limitations of the proposed approach is the additional burden for the framework developer, though the use of a documentation tool like *HiFi* can help to reduce this effort. Nevertheless, considering that developing a framework is a hard and time-consuming task, and the success of a framework is highly tied to its usability, producing good documentation can be considered an activity within the development process which is worth investing efforts.

7. Acknowledgements

The authors thank Analía Amandi and Roberto Moriyón for the invaluable discussions about this work and comments on the manuscript. They would also like to thank to anonymous referees of the SEKE'99 Conference for their comments on a preliminary version of this paper.

8. References

[1] Butler G., Dénommée P. Documenting Frameworks. In: Building Application Frameworks. Chapter 21. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, 1999. ISBN #0-471-24875-4

[2] Campo M., Price T. Luthier: Building Framework Visualization Tools. In: Implementing Application Frameworks. Chapter 11. M.Fayad, D.Schmidt, R.Johnson (eds.) John Wiley and Sons, N.Y, June 1999. ISBN #0-471-15012-0

[3] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1994.

[4] Helm R., Holland I., Gangopadhyay D. Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of OOPSLA'90, ACM/SIGPLAN, New York, 1990.

[5] Johnson R. Documenting frameworks using patterns. In Proceedings of OOPSLA'92. ACM/SIGPLAN, New York, 1992.

[6] Johnson P., Wilson S., Markopoulos P., Pycock J.. ADEPT, Advanced Design Environment for Prototyping with Task Models. Proceedings of INTERCHI'93, ACM Press.

[7] Lajoie R., Keller R. Design and reuse in object-oriented frameworks: Patterns, Contracts, and Motifs in Concert. In Object-Oriented Technology for Database and Software Systems. V.Alagar, R.Missaoui (eds.), World Scientific Publishing, Singapore, 1995.

[8] Meyer B. Applying Design by Contract. IEEE Computer, October 1992. Volume 25, Number 10.

[9] Ortigosa A., Campo M. SmartBooks: A Step Beyond Active-Cookbooks to aid in Framework Instantiation. In Technology of Object-Oriented Languages and Systems 25, June 1999, IEEE Press.

[10] Ortigosa A. Un método para la Aplicación de Documentación Inteligente en la Instanciación de Frameworks Orientados a Objetos. PhD Tesis, Universidad Autónoma de Madrid. February 2000. (in Spanish)

[11] Pangoli S., Paternò F. Automatic Generation of Task-oriented Help. Proceedings of UIST'95. ACM Press.

[12] Paternò, F. Understanding Task Model and User Interface Architecture Relationships, CNUCE Internal Report, December 1997.

[13] Pree W. Design Patterns for Object-Oriented Software Development. Addison-Wesley. 1994.

[14] Pree W. Framework Development and Reuse Support. In Visual Object-Oriented Programming, Concepts and Environments. M.Burnett, A.Goldberg, T.Lewis (eds.). Manning - Prentice Hall. 1995.
[15] Rumbaugh J., Jacobson I., Booch G. The UML Reference Manual. Addison-Wesley, 1999
[16] Richner T. Describing Framework Architectures: more than Design Patterns.
[17] Schappert A., Sommerland P., Pree W. Automated framework development. Symposium on Software Reusability (SSR'95), ACM Software Engineering Notes. Aug. 1995.
[18] Soundarajan N. Understanding Frameworks. In: Building Application Frameworks. Chapter 12. M.Fayad, D.Schmidt, R.Johnson (Eds.) John Wiley and Sons, N.Y, 1999. ISBN #0-471-24875-4
[19] Weld D. An Introduction to Least Commitment Planning. AI Magazine, Summer/Fall 1994.

[1] It is assumed that the framework documentation completely describes the functionality available. The user will not have any guide to implement functionality not described, even if it is provided by the framework.

[2] The solution adopted by UCPOP for providing universal quantifiers is to assign a type to every universally quantified variable. Every time one of these variables is found in a precondition, it is replaced by all the individuals of the corresponding type existing in the universe of discourse. The problem with this solution is that it assumes a static universe. This assumption is not true for the framework instantiation domain where elements like classes and methods are dynamically created during the instantiation process.